UNIVERSITÀ DI PISA

Progetto finale Laboratorio III

Angelo Quartarone, matricola: 598396

Introduzione

Il progetto consiste nella realizzazione di Hotelier, un servizio che implementa un sottoinsieme di funzionalità del popolare sito di recensioni (di alberghi, ristoranti e altri contenuti relativi ai viaggi), TripAdvisor.

Il progetto è implementato in Java, comprende sia lato server che lato client.

Albero delle directory

Di seguito una breve descrizione di come sono suddivise le directory del progetto per permettere una navigazione più semplice al lettore

All'interno della directory del progetto troviamo la seguente struttura:

- **bin**: questa directory contiene i file compilati in bytecode, pronti per essere eseguiti dalla Java Virtual Machine (JVM); è suddivisa ulteriormente in server e client, per mantenerne la separazione logica
- **lib**: contiene le librerie esterne necessarie per il funzionamento del progetto (in questo caso specifico contiene solo gson.jar, libreria utilizzata per il parsing dei file JSON)
- **src**: contiene tutto il codice sorgente, suddiviso in due sotto-directory server e client
- **Server.jar**: file eseguibile per avviare il server
- **Client.jar**: file eseguibile per avviare il client
- ulteriori file: troviamo infine i file JSON utili al server per gestire alcune funzionalità e permettere la persistenza dei dati
    - **Hotels.json**: contiene le informazioni base degli hotel utilizzati dal server
    - **Reviews.json**: contiene le informazioni delle recensioni per gli hotel
    - **Users.json**: contiene le informazioni degli utenti iscritti al servizio
    - **NOTA**: Se non si visualizzano i file Reviews.json e Users.json non allarmarsi, questi file vengono creati al primo avvio del server se non sono già presenti.

Di seguito si analizzeranno i singoli file del codice sorgente per permettere una profonda conoscenza del codice.

Codice sorgente

Server

Strutture Dati

Hotel.java

Questo file contiene la classe `Hotel` utilizzata per salvare tutte le informazioni delle strutture alberghiere.

Attributi

- `int id`: intero identificativo, ogni hotel ne ha uno unico
- `String name`: nome hotel
- `String description`: descrizione dell'hotel
- `String city`: città
- `String phone`: numero di telefono associato
- `ArrayList<String> services`: lista di stringhe che identificano i servizi offerti dalla struttura
- `int rate`: voto generale dell'hotel
- `Ratings ratings`: struttura dati `Ratings` contenete tutti i singoli voti ai servizi
- `int numReviews`: numero di reviews che l'hotel possiede

La classe offre il costruttore che prende in input le informazioni utili ad inizializzare tutti gli attributi, più una serie di metodi `get` per recuperare queste ultime.

Ratings.java

Qui è contenuta la classe `Ratings` che modella in una struttura unica l'insieme delle quattro aree per cui è possibile dare dei voti durante l'inserimento di una recensione

Attributi

- `int cleaning`
- `int position`
- `int services`
- `int quality`

In questa classe è presente il costruttore che istanzia l'oggetto con i valori in input e dei metodi di utility `get` per recuperare le informazioni

Review.java

In questo file è stata modellata la struttura dati utilizzata per gestire le recensioni nel calcolo local rank

Attributi

- `int rate`: voto generale
- `int cleaning`: voto pulizia
- `int position`: voto posizione
- `int services`: voto servizi
- `int quality`: voto qualità

- `String timeStamp`: timestamp utile a calcolare la recentezza della review

User.java

Qui troviamo la classe `User` che modella ogni singolo utente;

Attributi

- `public String username`: nome utente
- `private String password`: password
- `private static String filePath`: path al file Users.json
- `private int reviewCount`: numero di recensioni effettuate dall'utente

Metodi

Oltre il costruttore che prende in input nome utente e password, questa classe presenta dei metodi utili a gestire la struttura dati stessa:

- `private int loadReviewCount(String username)`: carica dal file degli utenti il numero di recensioni di un dato utente passato in input
- `private static List<User> getUsersFromFile()`: recupera dal JSON degli utenti le informazioni e le salva in una lista
- `private boolean checkPassword(String password)`: funzione utilizzata per controllare se la password inserita dall'utente è valida
- `public void insertUser(User newUser)` : metodo adibito all'inserimento di un nuovo utente quando quest'ultimo si vuole registrare sull'applicazione; legge le informazioni dal file Users.json, le aggiorna con l'oggetto User passato in input e le salva nuovamente
- `public boolean checkUser(User userToCheck)`: utilizzato in fase di login di un utente, controlla che le informazioni inserite siano corrette, ritorna true se queste sono giuste, false altrimenti
- `public static boolean checkUserName(String usernameToCheck)`: questo metodo viene utilizzato per evitare che due utenti differenti abbiano lo stesso username; prende in input la stringa contenente lo username da controllare e restituisce un booleano
- `public void addReviewPoints()`: aggiunge i punti assegnati ad ogni utente per ogni recensione aggiunta (utile poi al calcolo del badge personale), nello specifico, per ogni recensione vengono attribuiti 25 punti
- `public String getBadge()`: ritorna il badge assegnato all'utente, calcolandolo in base ai punti che quest'ultimo possiede
- `public String getUsername(), public int getReviewCount()`: metodi `get` che ritornano rispettivamente username e numero di recensioni

ServerMain.java

File contenente la classe principale che implementa un server che gestisce connessioni TCP da parte dei client e esegue un compito periodico. Appena il metodo main viene richiamato:

- fa partire un `Thread shutdown` che si occuperà della terminazione pulita del server nel caso cui venga volontariamente terminato tramite un ctrl+c
- chiama il metodo `init(String configFile)` che si occupa di recuperare tutte le informazioni dal file `serverParameters.properties` per inizializzare il server (indirizzi per connessioni TCP e UDP con relative porte, path al file Hotels.json)
- avvia il `SingleThreadScheduledExecutor()` che si occupa dell'aggiornamento periodico dei ratings degli hotel e dell'invio delle notifiche (che annunciano dei cambiamenti nel ranking di un hotel) ai client iscritti (tramite protocollo UDP)
- inizia un ciclo in cui accetta le connessioni in arrivo dai client, una volta accettata, quest'ultima viene passata ad un thread di una `CachedThreadPool` che se ne occuperà (se richiesto quindi dal carico di lavoro viene creato un nuovo thread, altrimenti viene riutilizzato un thread già esistente)

ScheduledTask.java

Questo file contiene la classe ScheduledTask (che implementa `Runnable`) contenente la logica del thread schedulato; il periodo con cui questi task vengono eseguiti è un input del programma (espresso in millisecondi).

il costruttore `public ScheduledTask(String hotelsPath, int udpPort, String udpIp)` prende in input:

- una stringa che identifica il path al file contenente gli hotel (utile al salvataggio delle informazioni)
- un intero che identifica la porta per la connessione UDP
- un intero che identifica l'indirizzo di multicast per la connessione UDP

Questo inizializza inoltre un `SearchEngine` e un `ReviewEngine` (di cui si troverà la descrizione più avanti) oltre che una `ConcurrentHashMap<String, Hotel>()` utilizzata come struttura di appoggio per eseguire i vari controlli.

L'unico metodo presente in questa classe è `public void run()` in cui, viene calcolata la media dei ratings tramite metodi esterni contenuti nella classe `ReviewEngine` e aggiorna il file Hotels.json; successivamente viene controllato se ci sono stati cambiamenti riguardanti il primo hotel in classifica (la mia personale implementazione di ranking prevede una classifica per ogni città e non una generale) e manda un messaggio UDP in multicast se trova differenze.

CommunicationManager.java

Questo file contiene una classe di utility; questa classe centralizza la comunicazione (che viene effettuata tramite `DataInputStream` e `DataOutputStream`) in un unico luogo, in modo da semplificare le azioni di invio e ricezione tramite due metodi: `send()` e `receive()`

SessionManager.java

Questo file contiene la logica dei thread che gestiscono i client. La classe `SessionManager` anch'essa che implementa `Runnable`, come dice lo stesso nome, contiene tutti i metodi utili a gestire tutte le azioni che un client può fare all'interno della stessa sessione

Attributi della classe:

- `private Socket socket`: client socket
- `private String hotelsPath`: path al file degli hotel
- `private CommunicationManager communication`: classe che si occupa della comunicazione
- `private State actualState`: stato attuale della sessione
- `private ConcurrentHashMap<String,LinkedBlockingQueue<Hotel>>hotels`: struttura dati utilizzata per contenere gli hotel, questa può essere vista come una cache, unica per ogni sessione
- `private SearchEngine searchEngine`: classe che si occupa della ricerca
- `private User actUser`: classe che implementa un utente

il costruttore `public SessionManager(Socket s, String hotelsP)` prende in input:

- la socket su cui è collegato un client
- una striga contenente il path al file Hotels.json

quello che fa il metodo `public void run()` è ciclare finchè la socket non risulta chiusa in modo da chiamare il metodo per gestire i messaggi ricevuti; qui di seguito sono presenti delle brevi descrizioni per ogni metodo presente nella classe:

Metodi

- `private boolean handleMessage()`:

  adibita a fare il parsing dei messaggi ricevuti del client in modo da chiamare il metodo associato ad ogni tipo di messaggio; questo parsing è diviso logicamente in due stati differenti, in base a se l'utente ha eseguito il login (la classe ha un attributo in cui è salvato lo stato attuale della sessione)

- `private void registerUser(CommunicationManager communication)`:

  prende in input un `CommunicationManager` e si occupa di gestire la fase di registrazione di un utente; da notare come non sia possibile l'utilizzo di uno stesso username per due utenti distinti

- `private void loginUser(CommunicationManager communication)`: prende in input un `CommunicationManager` e si occupa della fase di login; nel caso di login avvenuto con successo modifica lo stato attuale in "logged"

- `public void searchHotelByCity(CommunicationManager communication)`:

  prende in input un `CommunicationManager` e si occupa della ricerca di tutti gli hotel di una determinata città richiesta dal client; il flusso di esecuzione è il seguente:

  - chiede all'utente la città per cui vuole effettuare la ricerca
  - controlla se la chiave città è presente nella hashmap, se questa non è presente, viene caricata la lista di hotel di un'intera città
  - viene formattata la lista e inviata al client

- `public void searchHotel(CommunicationManager communication)`:

  prende in input un `CommunicationManager` e si occupa della ricerca di un hotel specifico. il flusso di esecuzione è il seguente:

  - chiede all'utente città e nome hotel
  - controlla se la chiave città è presente nella hashmap, se questa non è presente, viene caricata la lista di hotel di un'intera città
  - viene cercato lo specifico hotel all'interno della lista di hotel, formattato e inviato al client

    NOTA: anche quando viene ricercato un hotel specifico viene caricata in memoria tutta la città di cui quell'hotel fa parte, questa è una mia personale scelta implementativa che introduce una maggiore efficienza in caso di ricerca; esempio: se un utente ricerca un hotel specifico potrebbe volere cercare anche altri hotel della stessa città, per cui già si avrebbero i valori; al contrario, se un utente prima ricerca per città (con il metodo `searchHotelByCity`) e successivamente vuole cercare solo uno specifico hotel di quella stessa lista, essendo già presente in memoria, quest'ultimo non deve essere caricato nuovamente dalla memoria

- `public void addReview(CommunicationManager communication)`:

  prende in input un `CommunicationManager` e si occupa dell'aggiunta di una recensione per uno specifico hotel (per eseguire questa azione bisogna prima avere effettuato il login). All'utente sarà quindi richiesta città e nome hotel, se con queste informazioni non viene trovato l'hotel, il metodo fallisce informando l'utente che l'hotel non è stato trovato, al contrario sarà quindi richiesto di inserire le valutazioni per i campi: rate, cleaning, position, services e quality. A questo punto la recensione verrà caricata sul file JSON delle recensioni. A recensione completata vengono anche aggiunti all'utente dei punti che serviranno a calcolare il badge (livello di esperienza basato sul numero di recensioni)

- `public void badge(CommunicationManager communication)`:

  prende in input un `CommunicationManager` e si occupa di informare l'utente del badge che ha acquisito inserendo recensioni

- `private void exit(CommunicationManager communication)`:

  si occupa della chiusura della sessione e quindi della comunicazione

SearchEngine.java

Questo file contiene il motore di ricerca del server, ovvero tutti gli algoritmi utili alla ricerca di hotel e la manipolazione di questi.

il costruttore `public SearchEngine(String file)` prende in input la stringa del path al file degli hotel.

Metodi

NOTA: In questa classe è presente un largo uso di quella che ho deciso essere la struttura dati più consona per questo progetto; `ConcurrentHashMap<String, LinkedBlockingQueue<Hotel>>` è l'implementazione concorrente di una hashmap in java, in questo caso specifico utilizza come chiave una stringa (il nome di una città) e come valore una coda (versione concorrente di una coda in java) contenente tutti gli hotel della stessa città. Inoltre ritengo utile fare sapere che i prossimi 2 metodi sono gli **unici** utilizzati per leggere dal file principale degli hotel. È poi presente l'**unico** metodo per la scrittura su file ed altri metodi di utility.

- `public void updateHotelHashByCity(String cityFilter,ConcurrentHashMap<String, LinkedBlockingQueue<Hotel>> existingHotels)`:

  questo metodo si occupa della ricerca su file di tutti gli hotel appartenenti ad una data città. Prende in input: la stringa contenente la città da filtrare e la `ConcurrentHashMap` da aggiornare. `JsonReader` è la classe (offerta dalla libreria esterna gson) che permette di scorrere il JSON e quindi di recuperarne le informazioni. Il ciclo principale di questo metodo scorre ricorsivamente all'interno i campi del file estraendone con uno switch case tutte le informazioni. Infine, dopo aver creato un'oggetto `Hotel`, lo aggiunge alla coda della propria città, se questa non è presente, la crea.

- `public ConcurrentHashMap<String, LinkedBlockingQueue<Hotel>> getHotelsHashMap()`:

  questo metodo scorre il JSON come il precedente, con la differenza che non prende parametri in input, bensì restituisce una hashmap contenente tutti gli hotel del file.

- `public void saveHotelsHashMap(ConcurrentHashMap<String, LinkedBlockingQueue<Hotel>> hotelsMap, String filePath)`:

  come anticipato, questo è l'unico metodo a cui è affidato il compito di sovrascrivere le informazioni sul file Hotels.json quando avvengono determinati cambiamenti. Questo metodo prende in input la hasmap con gli hotel e la stringa che contiene il path al file, utilizza la classe `JsonWriter` offerta dalla libreria gson, iterando su tutti gli hotel della struttura dati sovrascrivendo di fatto i dati e mantenendo questi ultimi aggiornati.

- `public Hotel searchByHotelName(String cityFilter, String hotelName, ConcurrentHashMap<String, LinkedBlockingQueue<Hotel>> hotels)`:

  questo algoritmo è il responsabile della ricerca di un hotel all'interno della `ConcurrentHashMap`. Prende in input: la città, il nome dell'hotel e la hashmap in cui cercare; in questo caso, date le proprietà note di questa struttura dati, la ricerca risulta efficiente, rendendo di fatto l'applicazione reattiva nel caso di ricerche consecutive e pertinenti.

- `public String formatHotelsHash(ConcurrentHashMap<String, LinkedBlockingQueue<Hotel>> hotels)`, `public String formatHotelsList(LinkedBlockingQueue<Hotel> hotelsQueue)`, `public String formatSingleHotel(Hotel hotel)`:

questi tre metodi si occupano semplicemente di formattare le diverse strutture dati che contengono le informazioni, tutti e tre prendono in input la struttura interessata e ritornano una stringa formattata; vengono usati in tutti quei contesti di invio dati ai client, formattando le informazioni per renderle più piacevoli da visualizzare. È giusto tenere presente che il metodo per formattare l'intera hashmap non viene usato nel progetto, è stata usata per motivi di debug durante lo sviluppo.

- `public ConcurrentHashMap<String, Hotel> getBestHotelsMap()`:

  usata nel contesto in cui bisogna recuperare i migliori hotel per ogni città, questo metodo non prende input ma restituisce una `ConcurrentHashMap` con chiave "città" e valore `Hotel`

- `public String getChangedHotelsString(ConcurrentHashMap<String, Hotel> previousHotels, ConcurrentHashMap<String, Hotel> updatedHotels)`:

  l'ultimo metodo di questa classe centrale per il funzionamento dell'applicazione viene utilizzato, diversamente da prima, per formattare la notifica dei migliori hotel contenuti nella struttura. Ritorna la stringa da inviare

ReviewEngine.java

Questo file, insieme al precedente, risulta di particolare importanza per il funzionamento del programma.

Attributi:

- `private String reviewPath`: path al JSON utilizzato per salvare le informazioni delle recensioni degli hotel
- `private String hotelPath`: path al file degli hotel

Il costruttore `public ReviewEngine(String hotelPath)` inizializza semplicemente il path file

Metodi

- `public void addReview(int hotelIdentifier, int rate, int cleaning, int position, int services, int quality)`:

  il primo metodo, responsabile dell'inserimento di una recensione nel file, prende in input:

  - `int hotelIdentifier`, l'intero identificativo dell'hotel (in questa funzione viene usata un'altra `ConcurrentHashMap` con chiave identificatore dell'hotel e valore la lista delle recensioni)
  - `int rate`: rate generale dell'hotel
  - `int cleaning, int position, int services, int quality`: valori dei singoli campi di cui una recensione è composta

  Il metodo legge i dati tramite il metodo `fromJson` (offerto dalla libreria esterna) riempendo la hashmap, aggiunge la nuova review nella lista dell'hotel (aggiungendo anche un timestamp) e riscrive la struttura sul file usando `toJson`.

- `public ConcurrentHashMap<Integer, List<Review>> getReviews()`:

ritorna la hashmap con tutte le recensioni; non prende parametri in input

- `public ConcurrentHashMap<Integer, List<Review>> calculateMeanRatesById()`:

  NOTA: questo metodo è la mia personale interpretazione di un possibile algoritmo per il ranking di un hotel basato sulle recensioni. Sono presenti ulteriori commenti esplicativi nel codice sorgente, di seguito una breve spiegazione del suo funzionamento:

  non prende valori in input e ritorna una `ConcurrentHashMap<Integer, List<Review>>`, dopo aver caricato i dati dal file, calcola una media pesata in base alla recentezza della recensione per ogni campo della stessa. Questo meccanismo, unito all'intrinseco valore qualitativo delle *"stelle"* e al numero di recensioni, favorisce a creare il ranking locale (inteso per città) dell'applicazione. Per differenziare questa da altre implementazioni, ho quindi deciso di non calcolare un punteggio distillato da vari parametri, ma di calcolare una semplice media pesata sulla recentezza e discriminare il primo classificato in base al numero di recensioni

- `public void updateHotelFile(ConcurrentHashMap<Integer, List<Review>> reviewHashMap)`: l'ultimo metodo, utilizzato insieme al precedente permette di andare a salvare i valori delle recensioni sul file originale degli hotel, rendendo così le recensioni accessibili a tutti gli utenti. Il tipo in input è uguale al tipo di output del precedente metodo, permettendo così di usare in un'unica soluzione il calcolo delle medie e la scrittura su file di queste ultime aggiornate

Client

Lato client, troviamo una quantità di codice molto inferiore, poichè tutta la gestione della sessione, delle strutture dati e della logica è implementata sul server; l'unico compito del client è quindi quello di creare un layer di comunicazione tra l'utente finale e l'applicazione vera e propria, leggendo gli input da tastiera per inviarli e riceverne le informazioni o le azioni richieste.

Troviamo tre file:

ClientMain.java

In questo piccolo file che contiene la classe `ClientMain`, l'unica azione che troviamo è quella di istanziare un nuovo oggetto `HOTELIERCustomerClient` ed avviare la `Command Line Interface`

CommunicationManager.java

Questo file è il gemello lato client del `CommunicationManager`, di cui sono già state analizzate le funzionalità precedentemente

HOTELIERCustomerClient.java

Infine, il file contenente tutta la logica e i metodi utilizzati per favorire l'interazione utente-server.

Attributi

- `private String ipAddr`: indirizzo ip per comunicazione TCP
- `private String tcpPort`: porta TCP
- `private String udpPort`: porta UDP

- `private String udpIp`: indirizzo IP per ricezione pacchetti UDP di notifica
- `private Socket socket`: socket per la comunicazione TCP
- `private MulticastSocket multicastSocket`: socket per la comunicazione UDP
- `private List<String> receivedMessages`: lista che contiene i messaggi UDP ricevuti
- `private volatile boolean running`: variabile utilizzata per interrompere i cicli principali utilizzati nel codice

Metodi

Oltre il costruttore che non prende input ma che inizializza il client tramite la funzione `init`, troviamo i seguenti metodi:

- `private void init(String configFile)`: questo metodo prende in input il path al file di configurazione e si occupa di inizializzare tutti gli attributi della classe
- `public void runCLI()`: questo metodo è il cuore del client, si occupa di gestire la comunicazione con l'utente e di inviare i messaggi al server; il flusso di esecuzione è il seguente:
  - viene chiesto all'utente di inserire il comando
  - viene inviato al server che procederà con l'elaborazione
  - viene ricevuta la risposta dal server e viene stampata a video
  - il ciclo si ripete finchè l'utente non decide di uscire
- `public void startMulticastListener(String udpIp, String udpPort)`: questo metodo si occupa di avviare un thread che sta in ascolto dei messaggi UDP in multicast
- `public void stopMulticastListener()`: metodo che interrompe il thread che sta in ascolto dei messaggi UDP
- `public void printReceivedMessages()`: metodo che stampa a video i messaggi UDP ricevuti. Da notare che in questa classe non è presente una sincronizzazione esplicita per la gestione della CLI, questo perchè i messaggi UDP ricevuti in una sessione vengono salvati in una lista e stampati solo quando il server comunica che è possibile stamparli, in modo da evitare conflitti di stampa.

Diagramma Thread Server

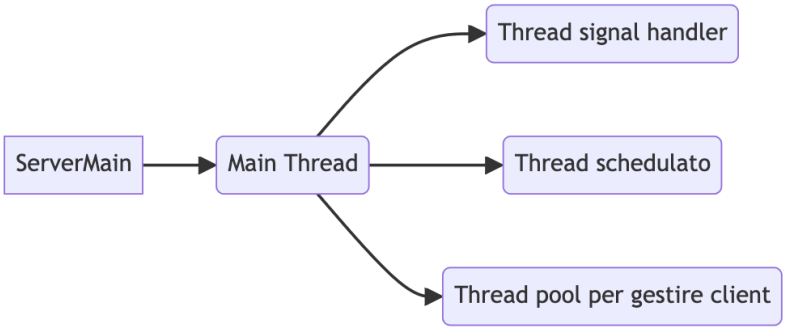

Diagramma Thread Client

Istruzioni per l'avvio

Una volta estratto il file .zip, il progetto può essere avviato nei seguenti modi:

- tramite l'uso del makefile:
  - `make server` per compilare il server, `make server_run` per avviare il server
  - `make client` per compilare il client, `make client_run` per avviare il client

  oppure

  - `make server_run_jar` per avviare il server tramite il file jar
  - `make client_run_jar` per avviare il client tramite il file jar

  NOTA: Nel caso di utilizzo del makefile, il server verrà avviato con un periodo di aggiornamento di 30 secondi, è possibile modificare questo valore direttamente nel makefile

- tramite file jar:

    - `java -jar Server.jar <Periodo in millisecondi>` per avviare il server
    - `java -jar Client.jar` per avviare il client

- manualmente:

    - `javac -cp ./lib/*.jar ./src/server/*.java -d bin` per compilare il server
    - `javac -cp ./lib/*.jar ./src/client/*.java -d bin` per compilare il client
    - `java -cp bin:lib/gson.jar server.ServerMain <Periodo in millisecondi>` per avviare il server
    - `java -cp bin:lib/gson.jar client.ClientMain` per avviare il client